A Framework for Distributed System Designs

Martin S. Feather
USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey CA 90292
email: feather@isi.edu

Stephen Fickas
Department of Computer Science
University of Oregon, Eugene OR 97403
email: fickas@cs.uoregon.edu

Abstract

We present a framework to structure the space of designs for a class of distributed systems. The purpose of this framework is to codify known design knowledge, and thus, when given the task of developing a new system in this class, to facilitate:

- *navigation - finding designs applicable to the task*
- *evaluation - identifying the strengths of weaknesses of a given design, comparing alternative designs*
- *reification - realizing an abstract design as the solution to a concrete task*

This is illustrated on the class of resource control systems operating in a distributed setting.

Keywords: *design, distributed systems, idealized requirements, resource control.*

1: Introduction

Our interest is in the design of 'similar' systems, that is, systems that fall into a given class (e.g., the class of resource management systems). In particular, we are interested in classes whose systems admit to a wide variety of designs - the wider the variety, the more need for support to codify design knowledge, so that such knowledge will be applicable during design of the task at hand.

Our goal is to make use of prior knowledge on how to design systems in the class. The idea of reuse of knowledge gained from past designs is, of course, well established. For example, the Draco work [12,8] is an early representative of the style in which design is done by refinement from a specification expressed in a domain-specific language. Another approach to reusability is to accumulate past design cases into a repository, from which case-based retrieval mechanisms are employed to select the design(s) closest to the specification of the current task (e.g., [13]). Our approach is to accumulate general design knowledge about a class of problems within a framework designed for that class. When faced with the task of designing another instance in the class, we expect our framework to support the following activities:

Navigation - as the designer navigates through the space of design knowledge, the organization of the framework should ease the designer's task of finding knowledge that is potentially relevant to the new system, and, of course, minimizing time wasted considering knowledge that is not relevant.

Evaluation - because of our focus upon classes of systems whose instances admit to a wide variety of designs, a major part of the design activity will concern the evaluation of candidate designs with respect to the needs of the task at hand. The framework should thus make available evaluation mechanisms, both to judge the strengths and weaknesses of a single design (with respect to the task requirements), and to compare and contrast alternative designs.

Reification - in accumulating design knowledge, there is the choice of whether to accumulate past cases of designs (data typical to a case-based reasoning approach, say), or whether to accumulate design principles gleaned from past cases of design. We follow the latter approach, and thus, given a specific task, the framework must provide mechanisms that apply the design principles to generate specific designs to meet that task.

We have chosen to focus upon *distributed* systems, and for such systems we advocate an approach based upon the following principles (which we think are relatively novel as a way of organizing design knowledge):

- begin from an expression of *idealized* requirements
- employ notions of constraints and responsibility to determine which components of the system will take action to meet the requirements
- emerge with a specification of interfaces as the means by which components exchange the information they need to live up to their responsibilities
- view the implementation of interfaces as the combination of three aspects: *when* information is transferred, between *whom* it is transferred, and *what* is the information being transferred.

Section 2 describes these elements in more detail. Section 3 presents our expression of a class of systems

(resource allocation systems) in this framework. Section 4 shows how a workshop problem fits into this class. Section 5 summarizes the status of our work and concludes the paper.

2: Elements of framework for distributed system designs

The key elements that form the basis of our framework are as follows:

Idealized requirements - the task requirements are first stated in an idealized, *non*-distributed fashion. It is in reference to these idealized requirements that many of the qualities of prospective designs can be judged.

Constraints and non-determinism - possible system behaviors are readily stated as the cross-combination of all possible behaviors of the individual system components (e.g., passengers entering/exiting elevators; elevators moving up/down floor by floor); requirements are expressed as constraints on these behaviors (e.g., passengers shall never be moved further from their destination), ruling out all those that fail to meet one or more requirement. The combination of these is a specification denoting precisely those of the possible behaviors that satisfy all the requirements.

Responsibility and constraint assignment - constraints are assigned as the responsibility of subsets of the components (e.g., passengers are responsible for entering elevators going their way). Only the components responsible for a constraint need to limit their actions so as to ensure its satisfaction (e.g., because passengers are responsible for entering elevators going their way, an elevator is free to open its doors at a floor at which there are passengers wishing to go in each direction, since only the appropriate passengers will board). Sometimes to do this responsibility assignment it is necessary to subdivide constraints into pieces such that their combination implies the original constraint, while each piece can be separately assigned as the responsibility of individual agents.

In the past we have made the argument for the importance of these concerns in doing system design [4]. Here, the novelty lies in the application of these ideas to a design framework, rather than to specific design tasks.

Interfaces - a component's responsibility for constraints induces a need for information by which to determine the appropriate actions to take (e.g., a passenger boarding an elevator will need to know in which direction that elevator is about to move in order to know whether he/she should in fact board that elevator). This need for information lies at the heart of the communication interfaces between components (e.g., lights and buttons in and around elevators). Note that it is completely unnecessary when requirements are stated in a non-distributed fashion, since there the assumption is that information is globally available.

Weakening of constraints - it will often be impossible (or deemed prohibitively expensive) to implement a perfect solution to the idealized requirements. This is particularly the case in a distributed setting. In reaction to this, it will often be necessary to weaken some of the constraints used to express requirements. Weakening a constraint allows violations of that constraint to occur; typically, weakening will introduce accompanying mechanisms, for example, 'repair' mechanisms that try to re-establish the violated constraint, or 'penalty' mechanisms that aim to discourage the violation of constraints in the first place. Such weakening may occur anywhere in the design process, whenever it is recognized that the current design is unrealizable.

3: Simple example - resource allocator

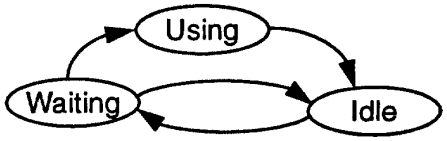
We use as a simple, illustrative example, that of a system to ensure the mutually exclusive use of a shared resource among a set of clients. An instance of this example was used as the common problem for consideration at the concurrency and distribution track at the 7th International Workshop on Software Specification and Design (IWSSD-7). We will show later how their problem statement is located in our design space.

In the style we are advocating, we explore the design space of solutions to this task as follows:

3.1: Idealized requirements, constraints and non-determinism

We begin by stating the requirements of this system as if it were not distributed, that is, assuming instantaneous access to a global dataspace of all information about the status of all parties. In particular, this simplifies from concerns of communication delays between distributed processes. Our statement of these idealized requirements is as follows:

Some number of clients exist, and may wish to use a shared resource. The possible behaviors of clients with respect to the resource are indicated in the following state-transition diagram:



Each client is in one of three mutually exhaustive states - Using (i.e, using the resource), Waiting (i.e., waiting to use the resource), or Idle (i.e., neither using the resource,

nor waiting to do so). Possible transitions between these states are shown as arrows in the above diagram.

The idealized requirements of this system are expressible as the following constraints:

C1) At most one client at once can be in the Using state.

C2) There should be no unnecessary waiting, that is, there should never be a client in the Waiting state and at the same time *no* client in the Using state.

C3) No client shall remain in the Waiting state longer than some (pre-determined) maximum period of time.

It is easy to formally specify the above behaviors and constraints in a model of system behavior comprising the state of the entire world (of clients, and the states they are in) and transitions between those states; all possible combinations of client activity produces a set of behaviors, where a behavior is an alternating sequence of states and transitions. Constraints become predicates on behaviors (constraints *C1* and *C2* can be evaluated with respect to individual states in those behaviors, while constraint *C3* needs access to the sequence of states in a behavior, and some measure of the passage of time). In a temporal model, they might be written as:

C1) $\Box\ \neg \exists\ c1\ c2\ .\ (\ \text{Using}(c1) \wedge \text{Using}(c2) \wedge c1 \neq c2\ )$

C2) $\Box \neg \exists\ c1\ .\ (\ \text{Waiting}(c1) \wedge \neg \exists\ c2.\ \text{Using}(c2)\ )$

C3) $\Box\ \forall\ c1\ .\ (\ \text{duration}(\text{Waiting}(c1)) < \text{max-wait-duration}\ )$

$\Box$ = always

The combination of possible behaviors and constraints denotes those and only those of the behaviors that satisfy all of the constraints.

3.2: Responsibility and constraint decomposition

The first step in our design model is to decide which components are 'responsible' for which constraints. Briefly, those and only those components declared as responsible for a constraint have to restrict their actions so as to ensure that constraint (in particular, *non*-responsible components are to be left free choice from among their possible actions). See [2] for details.

On the surface, it does not seem appropriate to make clients in the Idle state responsible for any of the above constraints. (In particular, a client should always have the choice of remaining Idle or transitioning to Waiting. Similarly, a client who is Waiting should not be required to, or prevented from, transitioning to Idle, and a client who is Using should not be prevented from transitioning to Idle.) We deliberately do not elevate this to the status of a hard and fast requirement, since we can imagine some circumstances in which the transition from Idle to Waiting could involve some mixed-initiative strategy between system and user, especially if we were to consider the states at finer levels of granularity. For the purposes of this paper, however, we will follow only the obvious (and most common) responsibility assignment that makes Waiting clients responsible for *C1* and *C2*, while both Using and Waiting clients are responsible for *C3*.

Note1: A reasonable alternative would be to have Using clients also share the responsibility for *C1* and/or *C2* (e.g., in pre-emptive scheduling, a Waiting client who is more important that a Using client is not to be kept waiting, and thus that Using client must cooperate by ceasing use of the resource - perhaps requiring an extension of the allowable client transitions to include the transition from Using back to Waiting).

Note2: An *un*reasonable alternative would be to have *only* Waiting clients responsible for *C3*, since the cooperation of Using clients to cease using the resource within some time bound is required (otherwise a client could 'hog' the resource, staying in the Using state while a Waiting client is kept from using the resource longer than the predetermined maximum waiting period). Since clients have no way of forcing each others' transitions, causing a Using client to cease using requires the cooperation of that client, i.e., sharing of responsibility.

3.3: Weakening of constraints

In a distributed setting, we may *not* assume that clients have instantaneous access to each others' status. Rather, they communicate only via channels that may introduce arbitrary but finite delays into the transmission of a message. For now, we will assume that other than such delays, transmission is perfect, that is, messages are never lost, garbled, or received in some order other than that in which they were transmitted.

Because of the potential transmission delays introduced by the nature of this distributed setting, we may need to weaken some of our idealized constraints. In particular, constraint *C3*, requiring that no client be kept waiting longer than some pre-determined period, may not be always attainable in many of the designs that we may wish to propose.

Similarly, since perfect synchronization between clients is unattainable (because of unpredictable transmission delay, and no global clock), constraints *C1* and *C2* also cannot (both) be guaranteed - consider the case in which there is one Using client, and one Waiting client; *C1* and *C2* require that as the Using client transitions to Idle, the Waiting client *simultaneously* transitions to Using. The reader might suspect that our mistake was to have got into the initial state of this scenario (a Using client who wants to transition to Idle, and a Waiting client); unfortunately, this cannot be precluded without limiting clients who are not responsible for the constraints (e.g., preventing Idle

clients from transitioning to Waiting), and/or precluding any Using whatsoever.

We thus face the need to *weaken* some or all of our requirements. By weakening, we mean that some states that violate the constraint *will* be allowed to occur, but we expect to emerge with a design that (tends to) minimize the frequency and duration of such occurrences.

In our example, we will be prepared to consider designs that weaken *C3*, and have the choice of whether to weaken one of both of constraints *C1* and *C2*:

W1) Weaken just *C2*, i.e., allow some 'unnecessary' waiting.

W2) Weaken just *C1*, i.e., allow multiple clients to be simultaneously Using.

W3) Weaken both *C1* and *C2*.

Choice *W1* means that we continue to require that at most one client is Using the resource at once, but will allow states in which 'unnecessary' waiting (i.e., in which there is no client currently Using, but there is some client Waiting) occurs. Even though such states are now to be allowed, we will compare alternative designs to see how they fare in terms of reducing the frequency and/or duration of this situation.

Choice *W2* is the opposite of choice 1; 'unnecessary' waiting is retained as a strict constraint, while multiple clients Using the resource at once is now to be allowed.

Airline reservation exhibits aspects of this form of weakening - up to a point, some overbooking of a flight is allowed (i.e., more clients may have reservations than the plane can accommodate) rather than (perhaps unnecessarily) denying them a reservation. Ultimately, the process of taking one's seats on the plane imposes a strict requirement, of course!

An intermediate choice is illustrated by a configuration management system for use within a close-knit group of software developers; when a developer wishes to edit some module but the system cannot rule out the possibility (e.g., because the network is down) that some other user is currently editing the same module, its design might be to allow the user to proceed with editing, and warn later if simultaneous editing (a violation of constraint *C1)* is thereafter discovered. This design eliminates unnecessary waiting at the cost of sometimes requiring developers to combine different edits of the same module.

We will restrict our attention in this paper to just the option *W1*, in which *C2*, but not *C1*, may be weakened.

3.4: Interfaces

In order for a component to ensure the constraints for which it is responsible, it must be able to distinguish those of its possible actions that are safe (i.e., lead to satisfaction of the constraints) from those that are unsafe (i.e., lead to violation of the constraints) and which therefore must not be taken. In a distributed setting it is typically the case that not all of the information a component needs to make these distinctions is 'local' to that component, and therefore it must obtain the needed non-local information from the other components. This *need for information* is the rationale for the interfaces between components, and how they are used [6].

Focussing on *C1*, which is:
$\Box \neg \exists$ c1 c2 . (Using(c1) $\wedge$ Using(c2) $\wedge$ c1 $\neq$ c2)
we can see that this implies the addition of the following necessary and sufficient precondition on the transition of client c from Waiting to Using:
$\neg \exists$ c1 . Using(c1) $\wedge$ c1 $\neq$ c
that is, there is no other client currently using the resource. Conditions such as this can be calculated given a constraint and a prospective action, by computing the weakest precondition of the action that will ensure the continued satisfaction of the constraint, and simplifying (see [3] for further details).

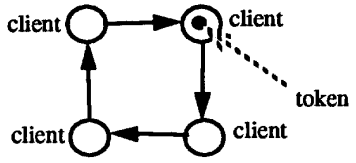
In our system, the local information of a client does *not* include the status of other clients, hence there is the need for an *interface* to provide this information.

The need to provide information serves as the rationale for an interface, but still leaves considerable choice in how to implement the interface, that is, how to provide the information. We identify three orthogonal aspects of such interface design:

- *when* to initiate the transfer of information
- between *whom* is information transferred
- *what* information is transferred

We give a simple example, and then examine each of these aspects in more detail:

Consider a design in which clients are organized into a ring, and there is a single token circulating around that ring, e.g.,



Clients in this design behave as follows: a Waiting client in possession of the token transitions to Using; an Idle client in possession of the token passes the token on the next client clockwise in the ring; and a Waiting client not in possession of the token remains Waiting. To be Using, a client must retain the token; no such client may remain Using for more than some pre-determined max-using-duration time. In terms of the aspects identified above:

- *when:* information - in the form of the token - is transferred whenever the client possessing it is in the Idle state.
- *whom:* information transfer is to the next client clockwise in the ring.
- *what:* the token represents the information that only the possessor of the token may be in the Using state.

Use of a token is a common protocol for providing mutual exclusion in distributed systems [14]. See [11] for an approach to ensuring that the system components adhere to this protocol.

When: choices of when to initiate transfer information range from 'supply' style solutions, in which information is passed as soon as it becomes available (i.e., the provider of the information is keeping the recipient as up-to-date as possible, modulo communication delays) to 'request' style solutions, in which the recipient must request information to trigger its transfer. This is reminiscent of forward- and backward-chaining derivation.

For example, a 'request' style token-ring would have the token stay put at an Idle client until request(s) for the token are received. Once requested, the token will then begin to circulate until it reaches the first Waiting client, say.

Whom: choices of between whom information is transferred include 'broadcast' styles, where everyone is involved, 'individual' styles, where transfer is between pairs of clients or hybrids of these. Transfer may be determined statically (ahead of time) or dynamically (based on run-time conditions).

For example, the ring architecture pictured earlier employs a static, individual style of transfer, where each client transfers the token to its immediate clockwise neighbor.

Alternatively, an architecture in which a Waiting client asks all the other clients whether they are currently Using the resource exhibits the broadcast style.

Finally, consider a central repository to which clients supply the token when they are not Using, and from which they request the token; in the case of competing such requests, is given to the client who is the least recent user of the resource. This demonstrates static individual transfer from clients to the repository, and dynamic individual transfer from repository to clients.

What: choices of what information is transferred depend upon whether, and how, the information required can be decomposed. In the resource allocator, the information needed by client c is the answer to: $\neg \exists$ c1 . Using(c1) $\wedge$ c1 $\neq$ c. At one extreme are 'atomic' solutions in which the answer to this whole question is transferred - the simple token solution above has this style, since possession of the token implies that no other client is currently Using. At the other extreme are 'aggregate' solutions in which every client transfers information about its own status, and the aggregation of this information is sufficient to provide the answer. Note that because of communication delays, receipt of information may be some time after its transmission. Hence many of the solutions require the transfer of not simply the status of the sender at the time the information is sent, but also some commitment to how that status will or will not change in the future. Furthermore, with many of these solutions there is the obvious problem of overlapping information transfers - because of communication delays, different transfers of information may be overlap in time. Care must be taken to design the protocols so that components 'in the middle of' one transfer react accordingly if they become involved in a second transfer. We give brief examples of these 'aggregate' style solutions:

A client, upon becoming Waiting, asks the following[1] of all other clients: *'Will you guarantee not to be Using for at least 10 minutes following your reply to this query?'*. If it receives 'yes' answers from all the clients, it may then conclude that no other client will be using the resource within 10 minutes of when it first *sent out* the query (assuming that the local clocks of all clients run at the same speed; if the guarantee is somewhat weaker than this, e.g., that for any pair of clients, their local clocks run at rates differing by no more than

1. Since a detailed exposition is beyond the scope of this paper, we simply state this in natural language, and ask the reader to believe that this could readily be formally stated in an appropriate temporal language.

10%, then the deduced time will be correspondingly less). If it receives one or more 'no' answers, it cannot assume that resource will be free, and has to re-ask the question. Upon receiving such a query, a client who is Idle must answer 'yes' (and therefore commit to not using the resource for the next 10 minutes of local time); a client who is Using must answer 'no'; the interesting question is how should a client who is Waiting respond, because this is the case of overlapping information transfer: if such a client always answers 'no', then deadlock will likely occur, as the two (or more) Waiting clients continue to re-ask the question of each other, always responding 'no' when they receive the other's query. Thus some sort of tie-breaker is necessary. Any universally agreed-upon tie-breaking scheme that linearly orders all clients would suffice.

A similar solution, that does not make use of relative clock rates, would be to ask all other clients: *'Will you guarantee not to be Using from the time you answer this question until you send out such a query and receive 'yes' answers from all clients?'*. Again, some sort of tie-breaker is necessary to handle overlapping queries.

3.5: Comparisons

The evaluation of designs is facilitated by consideration of the same framework concepts. We briefly illustrate this point on two of our aspects of interfaces:

When: comparison of 'supply' style vs. 'request' style solutions reveals that the former tend to lead to more communication in lightly-loaded systems (e.g., an unused token flowing round and round the ring of clients), but under some circumstances can reduce waiting time (since there is no need to wait for a request to reach the current holder of the token to initiate transfer).

What: a timed token is more complex to operate than a plain one (one that denotes its holder as the only client who can be Using), but is less vulnerable to client failure or tardiness - the system must wait potentially indefinitely for the client holding a plain token, but has a finite time bound on how long to wait for a timed token to cease to be valid.

3.6: Claims illustrated by resource control example

The example of resource control has, we suggest, illustrated the elements of the framework that we outlined in section 3. In particular:

Idealized requirements - the requirements *C1*, *C2* and *C3* of an idealized resource controller were concise and simple to state and understand.

Constraints and non-determinism - the formal statement of clients, and the idealized requirements placed on the system, were readily captured as a combination of the possible non-deterministic behaviors of clients (the state-transition diagram) and the temporal predicates on behaviors implied by that state-transition diagram.

Responsibility and constraint decomposition - different responsibility assignments serve to concisely characterize radically different styles of solutions (e.g., pre-emptive scheduling requires Using clients to share responsibility). To the extent that we developed our framework, the constraints required little in the way of decomposition.

Interfaces - the most striking result of our attempt to build the framework for this class of distributed systems is the flexibility and coverage provided by our categorization of interfaces in terms of the *when*, *whom* and *what* aspects of information transfer. Different combinations of possible choices from among these aspects appear to characterize a wide variety of solutions.

Weakening of constraints - as expected, we found it necessary to weaken the idealized requirements in order to emerge with a realizable set of constraints.

The above arguments and preceding descriptions have all been in general terms, and/or have drawn upon snippets of simple examples for illustration. In the next section we show how two specific instances of resource allocation problems are located within our design space.

4: Instances within our design space

We now show how a (third-party) description of a class of resource allocation problems is located within our design space. We will then briefly indicate how a particular commercial system exhibits the features discussed.

4.1: Cancellable resource allocator

The problem class description is that provided by J. Kramer and R. Kurki-Suonio for consideration at their concurrency and distribution track of the recent 7th International Workshop on Software Specification and Design. Their problem statement, and how it relates to our design space, is shown in Figure 1.

4.2: FLEXlm, a floating license manager

We have also looked in detail at a particular instance of a cancellable resource allocator, namely FLEXlm, a commercial license manager distributed as part of Solaris 1.0. Space limitations here permit only a superficial summary of our study.

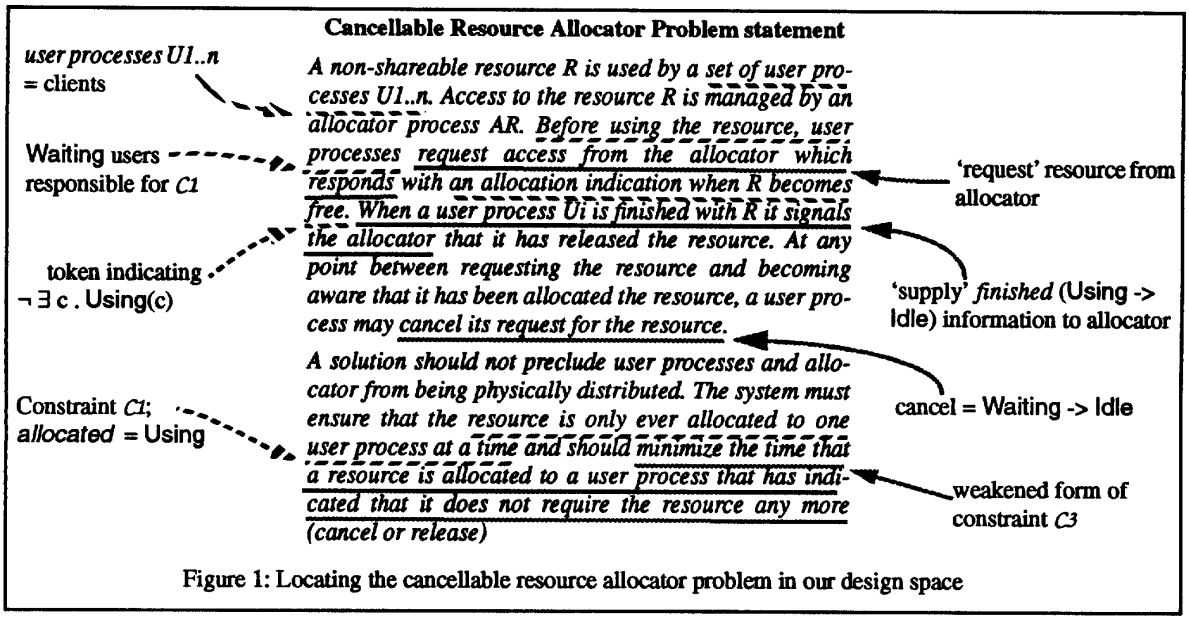


Figure 1: Locating the cancellable resource allocator problem in our design space

Briefly, FLEXlm permits a vendor of a piece of application software to ensure that within a network, the number of concurrent uses of the vendor's software never exceeds the number of licenses purchased to run that software (there are other possible restrictions that can be imposed, but we will not consider them here). This is a simple generalization of our constraint *C1* that permitted at most one user at once.

In its default configuration, for each application there is a single allocator program running on a node within the network, controlling user access to that application. This configuration of FLEXlm is thus an instance of our 'central repository' architecture.

To request use of the application, the would-be user contacts the allocator - i.e., in our terms, this is a 'request' style solution, and establishes a connection with the allocator across which it receives permission to run (or is denied permission, if there are already as many concurrent users as licenses purchased).

The application software checks its connection with the allocator at regular intervals. If the connection is found to be broken, and cannot be re-established, the application software will then halt. We recognize this as an instance of our simple 'timed token' style solutions of section 3.4. Each checking of the connection is equivalent to a renewal of the right to be using the resource.

There are many more intricacies of FLEXlm, for example, backup strategies for dealing with failure of the allocator. These too, we have found, can be justified in terms of design approximations of idealized requirements.

5: Conclusions

5.1: Current status of framework

At present, our framework is primarily a conceptualization, lacking the tool support required to make it readily applicable. Nevertheless, we are pleased with the insights that arise when simply following its principles as a mental discipline. We have developed some of the capabilities that would be required of a more complete implementation: [6] reports on representation and reasoning issues of this style of design; [3] on reasoning specifically about global and local constraints.

For the specific application of resource control, we have coded simulations on top of two base platforms, Regis [10] and AP5 [1] For Regis, we manually transformed a formal design of the FLEXlm system into two components: (1) A Regis "configuration specification" that establishes the distributed processes of the system. Regis hides the low level inter-process communication and the management of processes on separate hosts, leaving a high-level configuration language as our target. (2) A rule-based system that emulates a finite state machine for each process. The finite state machine provides the behavior of server and clients. For AP5, we built simple encapsulations of our 'supply' and 'request' style information transfer, such that instantiations of these automatically generate the appropriate protocol code for inter-client communication.

5.2: Navigation, evaluation and reification, and further work

Our original purposes in pursuing this framework were to facilitate the navigation, evaluation and reification of designs.

As reported herein, our studies of relatively small scale problems suggest that navigation is straightforward at least for these problems. We were able to locate both the workshop problem, and the FLEXlm license server, within our framework. The illustrations of section 3 showed that many variants are readily suggested by varying the elements of our approach (alternative responsibility assignments, constraint weakenings, and interface aspects). Our next step must be the application of this approach to more complex, larger-scale, systems.

By tying designs back to their idealized requirements, it becomes straightforward to estimate the ways in which a design may fail to fulfil its ideals. Furthermore, the relative strengths and weaknesses of designs can be judged by consideration of their respective potential for failure. Thus, we may generate test cases or scenarios *during design* that can be run in the production environment to highlight known problems [6,5]. For example, we can use design knowledge to produce a scenario that puts a load spike on a resource server, where such spikes are known to cause problems in existing client-server designs [7].

Reification requires the encoding of the design principles as reusable code (or, more likely, generators of code). Our preliminary experiments along these lines 'supply' and 'request' aspects of interfaces are encouraging, however much more work along these lines is needed.

5.3: Acknowledgments

The first author has benefited from the research context provided by ISI's Software Sciences Division and has been supported by Advanced Research Projects Agency contract No. F30602-93-C-0240. The authors wish to thank D. Tiktin for his work on the Regis-based experiment, Jeff Kramer and his group for Regis support, and Don Cohen for AP5 support. Views and conclusions in this document are those of the authors and should not be interpreted as representing the official opinion or policy of ARPA, the U.S. Government, or any other person or agency connected with them.

6: References

[1] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon.* SIGMOD RECORD 18(2), June 1989.

[2] M.S. Feather. Language Support for the Specification and Development of Composite Systems. *ACM TOPLAS*, 9(2):198-234, April 1987.

[3] M.S. Feather. Towards a Derivational Style of Distributed System Design. *Automated Software Engineering*, 1(1):31-59, March 1992.

[4] M.S. Feather, S. Fickas, and B.R. Helm. Composite system design: the good news and the bad news. In *Proceedings, Sixth Annual KBSE Conference, Syracuse, NY, October 1991*, pages 16-25. IEEE Computer Society, 1991.

[5] M.S. Feather, S. Fickas, and B.R. Helm. When things go wrong: predicting failure in multi-agent systems. In *Proceedings, Niagara Workshop on Intelligent Information Systems, Niagara, 1991*

[6] S. Fickas and R. Helm. Knowledge Representation and Reasoning in the Design of Composite Systems. *IEEE Transactions on Software Engineering*, 18(6):470-482, June 1992.

[7] S. Fickas and R. Helm. Automating Specification of Network Applications. In *Proceedings, IFIP International Conference on Upper Layer Protocols, Architectures and Applications, Vancouver, B.C., May, 1992.*

[8] P. Freeman. A Conceptual Analysis of the Draco Approach to Constructing Software Systems. *IEEE Transactions on Software Engineering*, SE-13(7):830-844, July 1987.

[9] R. Helm and S. Fickas. Scare Tactics: Evaluating Problem Decompositions Using Failure Scenarios In *Proceedings of the Workshop on Change of Representation and Problem Reformulation (Asilomar, CA, April 1992).* Technical Report FIA-92-06, NASA Ames Research Center, Moffett Field, CA 94025, pages 85-93.

[10] J. Kramer. *Regis Users Manual.* Department of Computing, Imperial College, London.

[11] N.H. Minsky. The Imposition of Protocols Over Open Distributed Systems. *IEEE Transactions on Software Engineering*, 17(2):183-195, February 1991

[12] J.M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, SE-10(5):564-574, September 1984.

[13] M. Pearce, A. Goel, J. Kolodner, C. Zimring, L. Sentosa, and R. Billington. Case-Based Design Support. *IEEE Expert*, 7(5):14-20,October, 1992.

[14] M. Raynal. *Distributed Algorithms and Protocols.* New York: Wiley, 1988.